SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Experiences with Poker ✓

5. TYPE OF REPORT & PERIOD COVERED

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
David Notkin, D. Socha, L. Snyder, M. Bailey, ✓
B. Forstall, K. Gates, R. Greenlaw, W. Griswold, T. Holman, R. Korry, G. Lasswell, R. Mitchell, P. Nelson

8. CONTRACT OR GRANT NUMBER(s)
N00014-86-K-0264 ✓

9. PERFORMING ORGANIZATION NAME AND ADDRESS
University of Washington
Department of Computer Science
Seattle, Washington 98195

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Information Systems Program
Arlington, VA 22217

12. REPORT DATE
April 1988

13. NUMBER OF PAGES
11

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of this paper is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
parallel programming, programming environments, language, algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
Experience from over five years of building nonshared memory parallel programs using the Poker Parallel Programming Environment has positioned us to evaluate our approach to defining and developing parallel programs. This paper presents the more significant results of our evaluation of Poker. The evaluation is driving our next effort in parallel programming environments; many of the results should be sufficiently general to apply to other related efforts.

DD FORM 1473 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

To appear in ACM SIGPLAN Symposium on Parallel Programming:
Experience with Applications, Languages and Systems
New Haven, Connecticut, July 19-21, 1988

Experiences with Poker*

David Notkin, David Socha, Lawrence Snyder,
Mary L. Bailey, Bruce Forstall, Kevin Gates, Ray Greenlaw,
William G. Griswold, Thomas J. Holman, Richard Korry,
Gemini Lasswell, Robert Mitchell, Philip A. Nelson

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Abstract

Experience from over five years of building nonshared memory parallel programs using the Poker Parallel Programming Environment has positioned us to evaluate our approach to defining and developing parallel programs. This paper presents the more significant results of our evaluation of Poker. The evaluation is driving our next effort in parallel programming environment; many of the results should be sufficiently general to apply to other related efforts.

1 Introduction

The Poker Parallel Programming Environment [17,18] eases the difficult task of programming parallel computers. Poker's development began in January 1982; the first public release took place in October 1985. To date, over 50 sites–including the University of Massachusetts, the University of California San Diego, and Intel Corporation–have versions of Poker. We are currently preparing version 4.1 for release. A large number of parallel programs have been implemented using Poker. Table 1 lists a few of the programs written at the University of Washington.

Table 1: Programs Built Using Poker

Cholesky decomposition [6]	FFT
Dynamic programming	SIMPLE [4]
Matrix multiply (systolic)	ADI
band matrix multiply	SOR
(systolic)	Polygon clipping
Vector-matrix multiply	WAP (systolic)
Matrix multiply [12]	LU-decomposition
(divide & conquer)	Transitive closure
Topological sort	Batcher's sort
Conjugate gradient	Jacobi iteration
Sharks & fishes [5]	Game of Life
Dataflow simulator	

The experiences gained from developing parallel programs with Poker has given us a chance to evaluate the strengths and weakness of the system. The goal of this paper is to present our evaluation, covering both conceptual and implementation issues. We hope our reflections guide other similar research and development efforts towards more fruitful approaches and away from other less promising ones. We focus on the most significant points of our complete evaluation. The full evaluation has driven the development of the requirements for *Orca*–our new parallel programming environment research effort.

Before proceeding, a few details about Poker are in order.

- Poker focuses on MIMD non-shared memory message-passing architectures. Further, the

*This research funded in part by Office of Naval Research Contract N00014-86-K-0264, National Science Foundation Grant CCR-8416878 and Air Force Office of Scientific Research Contract 88-0023. K. Gates is with the Department of Applied Mathematics, University of Washington, Seattle, WA 98195: P.A. Nelson is with the Computer Science Department, Western Washington University, Bellingham, WA 98225.

environment and programmers assume reliable message passing and reliable processing elements (PEs); our research is not concerned with fault-tolerance.

- Poker is based on the assumption that parallel programming is most effective when programmers handle parallelism explicitly [16]. Explicit parallelism has several key benefits. First, for many domains, programmers can develop parallel solutions to problems more easily than they can sequential solutions; for instance, concurrent programming languages have made operating system design more effective and reliable. Second, explicit parallelism does not hide execution costs from the programmer. Third, on the hardware side, explicit parallelism makes it much easier to create efficient executable code [16].

  The emphasis on explicit parallelism differs from other efforts where programmers write sequential programs and let compilers infer parallelism from the program [1,10]. One key argument made in favor of inferring parallelism is that it increases the potential for portability. As we discuss later, significant progress has been made in porting explicitly parallel programs.

- The original Poker system was developed for a specific architecture–the CHiP [19]. Several artifacts of this architecture are still visible in the current system, even though Poker now supports a broader class of architectures.

- The development of Poker began in early 1982, when the available technology was significantly less powerful than it is now. Most noticeably, workstations were not generally available. Hence, many of the design and implementation decisions that were appropriate then no longer are.

- As with many research projects, our current system has evolved from the original prototype. The prototype was constructed in one summer by about a dozen graduate students, and perhaps another dozen have worked on it since. Hence, it is no surprise that some of our problems arise from the lack of structure in the current implementation.

2 A Brief Overview of Poker

A common way to visualize a parallel algorithm is as a finite graph. For example, we usually view the systolic algorithm for matrix multiplication using a hex-connected mesh [11], the parallel FFT using a butterfly graph [3], and the tournament maximum finding algorithm using a binary tree [14]. In these *communication graphs*, the vertices represent processes and the edges represent the communication channels over which information is transferred between processes. Poker is an environment that lets programmers develop their parallel programs using a visual representation of the communication graph as the centerpiece. Just as this is a natural way to describe parallel programs, it is also a natural way to develop, debug, and maintain them.

A Poker program is implemented as a database rather than as a monolithic piece of symbolic text. The database contents are defined by three pictures, a set of program sources, and a table. The Poker programmer creates, manipulates, and sees the program through different *views*, or synthetic pictures of a specific type of information about the program. The views that support development of Poker programs are closely related to the elements of the database.

1. The *communication graph* is specified visually, using the SWITCH SETTINGS VIEW, and defines the structure of the computation.

2. The *process definition* is specified textually, using conventional editors like EMACS, and defines the computational activity for each of the separate PEs.

3. The *process labeling* is specified graphically, using the CODE NAMES VIEW, and defines which process each vertex will execute.

4. The *port name labeling* is specified graphically, using the PORT NAMES VIEW, and defines the names of the internal edges of the communication graph.

5. The *stream labeling* is specified tabularly, using the I/O NAMES VIEW, and defines the names of the dangling edges of the communication graph, thus supporting access to data files.

Since realistic Poker programs are too large to be easily described here, we present a Poker program that solves an extremely simple problem.

Example: Evaluate three univariate polynomials whose coefficients are given in three input streams; each polynomial is to be evaluated at a (possibly) different point given in the input, and the sum of the three results is to be returned in a (one element) output stream.

We solve this problem using a master-slave technique. The master reads the three points at which to evaluate the polynomials. The master sends the points to the slaves, each of which in parallel reads in a coefficient stream for one of the polynomials, evaluates it, and returns the result. The master, who waits while the slaves work, then sums the values and outputs the final result. The communication graph has a master vertex connected to three slave vertices. Each vertex has a dangling edge for a stream: each slave needs an input stream for coefficients, and the master needs a stream for the output.

In the Switch Settings View (see Figure 1), the boxes represent PEs and the lines represent datapaths. Using a mouse, the programmer connects PEs via datapaths by drawing lines between the PE boxes. The small black boxes on the perimeter, which terminate the diagonal lines, are called *pads*; they represent the sources or sinks of input or output streams. The name of the view is a vestige of the CHiP architecture, in which programmable switches selected the interconnection of the PEs.

Using the Code Names View (see Figure 2), the programmer assigns process names and parameter values to windows within each of the boxes. The **mastr** process is assigned to PE 1,1. It has three actual parameters (the points at which to evaluate the polynomials) and communicates with each of the other processes. The other three processes are instances of **slave**, each with a single parameter designating how many coefficients are to be read. The diagonal lines around the perimeter indicate which processes send data to or receive data from streams.

In the Port Names View (see Figure 3), the user names the datapaths using a representation similar to that of the Code Names View; the only difference is that each box contains windows corresponding to the eight compass points at which datapaths could connect to the PE. The programmer assigns the names used in the process definition to refer to the (logically) adjacent processes. The **mastr** uses the names **slav1**, **slav2**, and **slav3** for the slaves and **rpt** for the output stream; the slave ports are assigned to E,

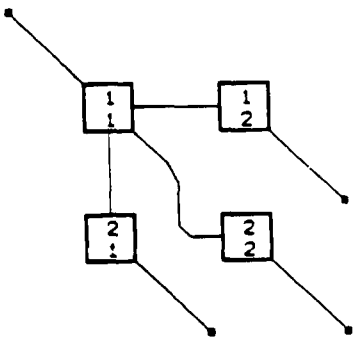

Figure 1: Switch Setting View

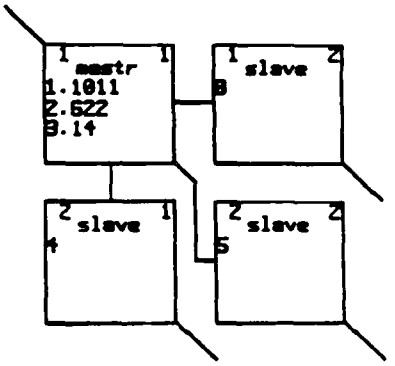

Figure 2: Code Names View

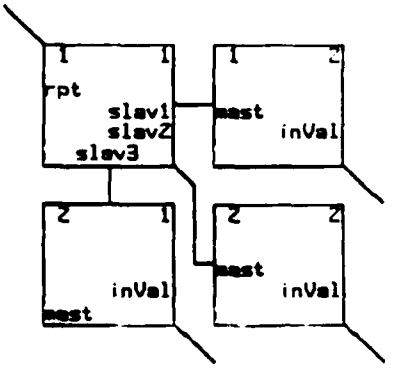

Figure 3: Port Names View

STREAM				DESTINATION				
PAD	NAME	INDEX	DIR.	PORT NAME	DIRECTION	CODE NAME	I	J
1	inSequence	1	input	inVal	southeast	slave	1	2
2	inSequence	2	input	inVal	southeast	slave	2	2
3	inSequence	3	input	inVal	southeast	slave	2	1
4	outValue	1	output	rpt	northwest	mastr	1	1

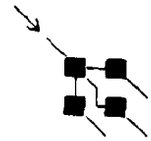

Figure 4: I/O Names View

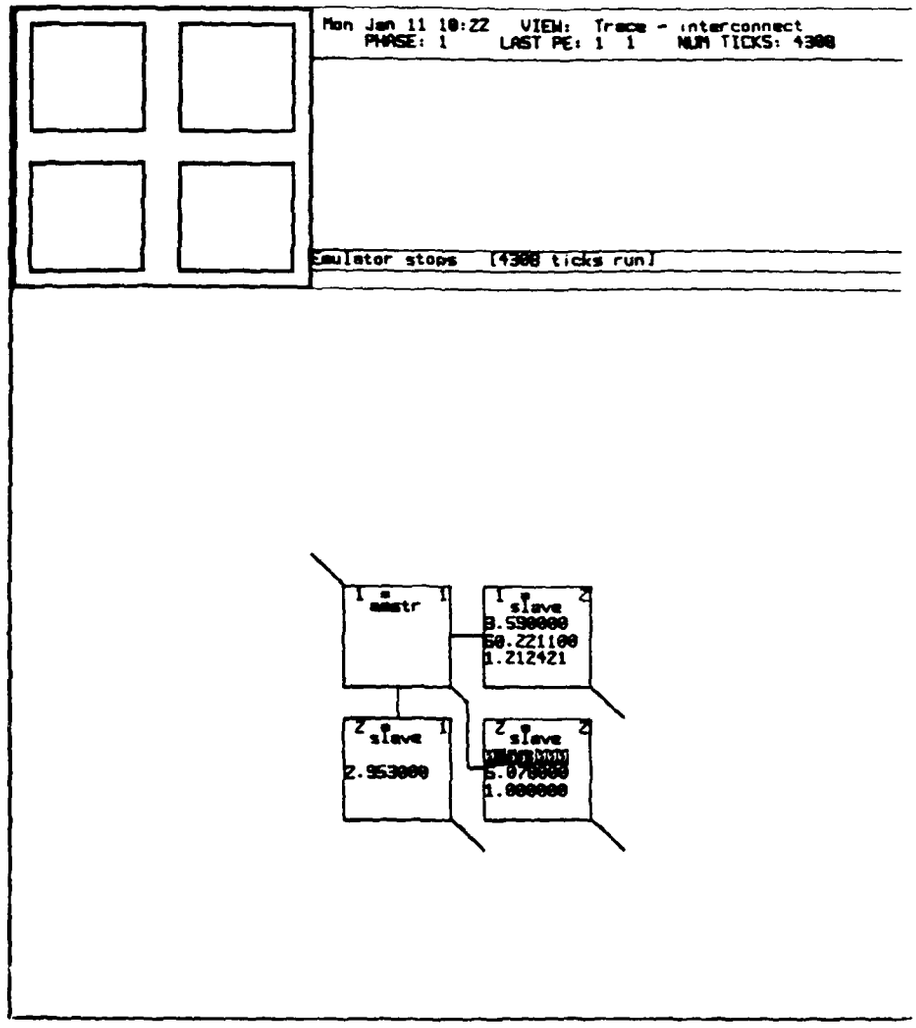

Figure 5: Trace View

SE, and S, while the output port is assigned to NW (see Figures 6 and 7). Port naming permits different processes to refer to the same channel with different names–the ends, not the datapath, are assigned names.

In the IO Names View (see Figure 4) the programmer specifies the stream information for each pad: the name of the file on the host operating system, the stream index, and whether the values are to be read or written. The system provides, as context, the destination information to the right of the vertical line.

The two text files representing the source code (see Figures 6 and 7) of the sequential processes and the input files (see Figure 8) are defined using conventional techniques. In this example, the source is written in the Poker C language. Poker C is conventional, except for the arrow assignment statements implementing interprocess communication. The names used for the `ports` declaration correspond to those mentioned in the Port Names View.

The four views plus the text files constitute a complete Poker program. Programs are converted to object form by compiling and assembling the database and processes. The object codes can then be loaded into a parallel computer or emulator. The execution of a Poker program can be watched using the Trace View (see Figure 5). As the program runs the current values of the variables specified in the `trace` declaration of the process specification (see Figures 6 and 7) are shown on the screen and highlighted as they change. This is useful for debugging and for watching dataflow patterns [20].

3 Evaluation

The Poker program just described provides a framework for presenting our evaluation. This section presents the most fundamental points from our complete evaluation.

3.1 Algorithm Decomposition

Our example focused on two levels of a Poker program. The $\mathcal{X}$-level, represented by the Poker C programs for `mastr` and `slave`, is for programming the sequential processes. It is conventional, sequential programming, with the addition of operations to read from and write to ports. The $\mathcal{Y}$-level, represented by the master-slave relationship, is for programming the

```
code mastr;
trace y1, y2, y3, result;
ports rpt, slav1, slav2, slav3;

main(x1, x2, x3) float x1, x2, x3;
{
  float y1, y2, y3, result;

  slav1 <- x1; /* Send point values */
  slav2 <- x2; /*   to slave PEs.   */
  slav3 <- x3;

  y1 <- slav1; /* Receive evaluated */
  y2 <- slav2; /*   polynomials     */
  y3 <- slav3; /*   back.           */

  result = y1 + y2 + y3;
  rpt <- result;
}
```

Figure 6: Poker C: `master`

```
code slave;
trace data, accum, xPower;
ports mast, inVal;

main(n) int n;
{
  int i;
  float data, accum, xPower, x;

  accum <- inVal;
  x <- mast;
  xPower = 1;
  for (i = 2; i <= n; i++) {
    data <- inVal;
    xPower = xPower * x;
    accum = accum + data * xPower;
  }
  mast <- accum;
}
```

Figure 7: Poker C: `slave`

```
1.004   ,5.078    ,2.953   ,
3.590   ,13.422   ,0.875   ,
6.000   ,1.444    ,0       ,
0.096   ,0        ,0       ,
```

Figure 8: An Input Stream (file `inSequence`)

communication graph. Visually defining this graph is perhaps the most novel feature of the Poker system. Together a $\mathcal{Y}$-level graph and a set of $\mathcal{X}$-level programs form a *phase*, corresponding to a logical activity of a parallel algorithm.

These two levels are not sufficient to solve complex problems. The polynomial example, for instance, must be combined with other phases to perform useful computation. The $\mathcal{Z}$-level supports composing phases to form a complete program. The composition may be a simple sequential ordering of phases, such as input, compute, and output, or it may require iteration and conditional checking, often to see if the results have reached sufficient accuracy.

The conceptual distinction among the $\mathcal{X}$-, $\mathcal{Y}$-, and $\mathcal{Z}$-levels has proven to be a powerful methodological tool that encourages effective parallel programming. Poker programmers find that physical problems naturally decompose into independent phases defined by graphs, and that these graphs typically are instances of one of a small number of graph families such as binary trees, meshes, and tori. Phases act as independent units of abstraction and execution analogous to procedures in sequential languages. Using the $\mathcal{Z}$-level language to define the flow of control among the phases increases flexibility in defining programs, in composing phases, and in reusing phases from other Poker programs. The three levels also give a high degree of modularity and reusability to Poker programs. Both phases and interconnections are reusable in different $\mathcal{Z}$-level or $\mathcal{Y}$-level programs. Similarly, Poker supports multiple $\mathcal{X}$-level languages, currently XX (a simple language without procedures) and Poker C, making it possible to select a language appropriate to the problem at hand. Other $\mathcal{X}$-level languages, such Lisp, are possible, although none has been implemented.

In Poker, the design and implementation of the $\mathcal{Z}$-level language are extremely rudimentary. Typically, the user replaces the $\mathcal{Z}$-level program by interactively executing each phase, watching it execute, and then progressing to the next phase. Although far short of a full-scale $\mathcal{Z}$-level language, a textual description of this interaction, along with conditional statements, can be fed into Poker. As our programs are becoming more complex and sophisticated, we require a more sophisticated $\mathcal{Z}$-level language.

Another drawback of the current $\mathcal{Z}$-level is that execution is required to synchronize at the end of each phase. (This is in part another artifact of the CHiP architecture, which changes the interconnection of the entire machine between phases.) Although some programs benefit from this approach, removing extraneous synchronization may substantially increase performance when running on architectures, such as n-cubes, that support complete connectivity [6]. So, Poker needs a more flexible $\mathcal{Z}$-level synchronization mechanism to support both styles of programs efficiently.

3.2 Visual Programming

The $\mathcal{Y}$-level environment is an instance of visual programming [7]. For documentation and explanation purposes, visual programming of communication graphs is an unqualified success. There are two drawbacks. One, creating large graphs by drawing each datapath can be tedious. Two, seeing patterns in large graphs is difficult due to the number of datapaths and crossings.

These difficulties arise in part because we do not effectively handle graph families. For instance, rather than individually building a 3×3 mesh, a 4×4 mesh, a 5×5 mesh, and so on, one would prefer to build a description to generate these different size meshes. Bailey and Cuny [2] show how graph grammars can help solve this problem. It is important to solve this problem while retaining the visual benefits. One idea we are exploring in *Orca* is to use programs to effectively generate the desired graph families.

3.3 Program Execution

Poker programs can be executed on an actual parallel computer, on a strict emulation of a parallel computer, or on a high-level simulator, with different speed and perhaps different function in each model. An emulator behaves exactly as the associated architecture, except for the elapsed time; relative costs of individual computations are identical. A simulator represents a "generic" parallel computer, but provides no guarantees about the degree to which it accurately models any specific architecture.

Actual computers are fast, but inflexible and costly in real dollars. Alternatively, our high-level simulator uses a light-weight process system to provide speed and versatility at the expense of exact timings. Using the simulator to design and debug has turned out to be a major advantage as it allows the user a great deal of control over the execution of the program and access to its run-time state [20]. However, even the simulator is not flexible enough to merge easily with a

fully interactive debugger. To achieve this flexibility in *Orca*, we plan to include an interpreter that walks the internal data structures directly.

3.4 Retargetability and Portability

Snyder and Socha [15] have shown that Poker can be retargetted to produce efficient code for different types of parallel architectures. Retargetting is possible because Poker assumes little of the underlying architecture: message-passing, MIMD execution, and an $\mathcal{X}$-level language for the target PEs. For most architectures that meet these requirements it is both conceptually easy and reasonable in practice to create efficient code from Poker programs.

Retargetability allows the Poker programmer to develop a program independent of the target architecture. The simulator, which has the most flexible debugging support, is generally used to test the program for its algorithmic correctness. When the program is working, the user can select another back-end, from within Poker, and run the program on the actual parallel computer *without changing any of the code.* The front-end and back-ends use a well-defined interface language, so it is easy to substitute one back-end for another and view the trace of the execution in the same Poker Trace View.

This program-architecture independence has its costs. Most notably, there is no way to access special features of the hardware from within Poker. The ability to specify optimizations for particular target architectures in such a way that the optimizations could be automatically incorporated into the program is needed.

3.5 Timing Model

Another major advantage to the simulator is that it uses a timing model for calculating the execution time of a Poker program. The user can specify the cost of Poker C operators, C control structures, and message packing, transmittal, and unpacking. While versatile enough to simulate different PE architectures, such as Transputers, the simulator does not model message forwarding or other communication costs [8]. Modified versions of the simulator can model costs such as the effects of communication co-processors [9].

3.6 Boundary Conditions

When a regular communication graph, such as a mesh, has one or more edges where processor connections are different from elsewhere in the graph (usually in the form of dangling communication lines), that graph has a boundary condition. We have identified two types of boundary conditions. In the first, I/O boundary conditions, the boundary is actually where data is input and output from the graph. This type occurs most often in systolic arrays, and is of interest here because the raw data must often be changed, for example, by padding the data with zeros or by adding additional information. The second type of boundary conditions, computational boundary conditions, occurs when edges bound a "computational space." In this case there must be some special code for dealing with the edges that specifies the values processors on the edge of the space would receive if the processors outside the space existed.

Boundary conditions inevitably complicate an algorithm implementation. Boundary conditions are usually handled in one of several ways. If the values needed from the edges are constants or are precomputable, the values can be entered as input to the processor array. I/O boundary conditions are often computed this way. Another possibility is to add processors that compute the values on the boundary and send them to the processors on the edge. A third possibility is to write special purpose code to handle boundary conditions. This code executes only on the edge processors, and calculates the input values, instead of actually receiving them from the external world.

The first approach keeps the programmer from writing special purpose code. However, it is slower, as it requires reads and writes on the edge processors that, in many cases, can be replaced by assignments. In addition, if the input values are precomputed, it usually makes more sense to do the computation as part of the algorithm's execution. Poker supports this approach reasonably well, except for the limitations of the I/O mechanisms mentioned in Section 3.8.

Using boundary condition processors leads to modularization of the special purpose code. It does, however, require extra communication, as well as using up a relatively large number of processors. In Poker, it has the additional disadvantage of increasing the amount of special purpose code. This happens because getting the data to the boundary condition processors requires that the internal processors must

add extra ports and send statements. A mechanism for providing some form of $\mathcal{Y}$-level fanout could help solve this problem, as the data used in computing the boundary values is usually the same data that is being shared between processors.

Poker gives programmers two ways to place special purpose code in the edge processors. The first is to handle them with $\mathcal{X}$-level statements that execute conditionally based on the PE location. The second option is to use compiler directives to create multiple programs with not quite identical code. The first approach is simpler for the programmer, but it produces bigger and slower programs; since PE memory is limited and the goal of parallel processing is efficient execution, this approach is unsatisfactory. In addition, there is some question whether or not a process should be guaranteed to know its position in the graph. The second approach is awkward, produces confusing source files with multiple sections of similar code, and unreasonably forces the user to maintain consistency among these variants. This is an area where the environment, with its knowledge of both the interconnection scheme and the code, could simplify the program, increase reliability, and decrease memory and cycle consumption of the underlying PEs.

3.7 Program Database

The visual, tabular, and textual views that represent a Poker program are also used when storing the programs. We store the information in a database, with a separate portion for the information from each of the views. Keeping these parts separate and having the views or compiler merge them as necessary is conceptually clean, especially for views that need only partial information.

Poker's implementation does not use a true database, which decreases the conceptual clarity and usability of the environment itself. The program parts are in separate files and the definitions of the access routines are scattered throughout Poker. The user must ensure that the database is not corrupted.

3.8 I/O

Messages to and from I/O pads may contain `bool`, `char`, `int`, and `float` values. This contrasts with messages between PEs which may contain arrays and structures as well. This restriction on I/O arises because of the way we store data streams in files. Each column contains a stream of data values for one pad. Multiple streams fit side by side in a single file so that the i^{th} row has the i^{th} element of each stream. Data values are of fixed width to allow easy access to the values as different PEs consume streams at different rates. This implementation causes two problems: files become difficult to handle manually, and PEs and pads can only exchange values of fundamental types.

In *Orca*, we plan to extend the full power of PE-to-PE communication to include PE-to-pad communication by allowing pads to read/write arrays and structures. The user will be able to view and create data files from an I/O editor similar to a spreadsheet. Each column represents a stream and each row an item in the streams. Cells can contain arbitrary data, perhaps including functions to generate values. This will simplify I/O programming and provide another debugging tool where the parallel I/O spreadsheet is used to watch the consumption and production of I/O values.

3.9 Environment Integration

Environmental integration was a key objective in the development of Poker. Integration is needed to aid the user in coping with the large amount of information in a parallel program. Poker's environment integrates multiple visual views to provide the user with a consistent and unified interface for visual programming and execution. This consistency, based on the communication graph, is a major advantage in programming and viewing the execution of Poker programs. The environment also gathers information from the program database and the converting this information into executables for a range of parallel simulators, emulators, and actual machines. Finally, the environment automatically starts and controls the execution of the program, providing the same interface for all of the back-ends.

Figure 9 shows the coarse structure of the parts of the Poker environment. The environment knows about everything above the dashed line. The programmer uses the views of the front-end to define the program (which is stored in a database), compile it, and trace it. A filled circle indicates read-write access; an open circle is read-only. The environment compiles the database into code for any of the back-ends on the right. During execution, control and trace messages flow between the front-end and back-end.

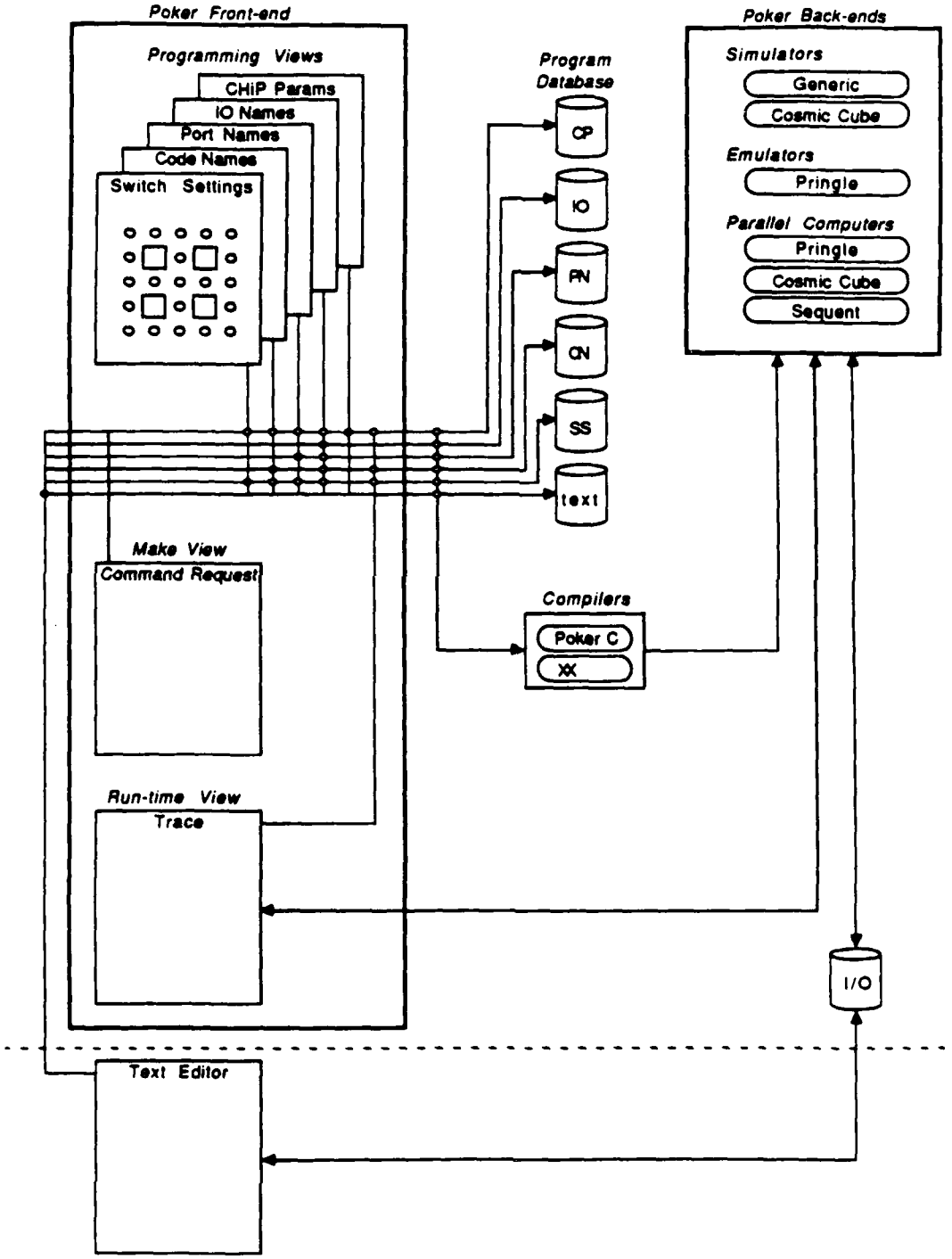

Figure 9: The Structure of Poker

3.10 Logical Graphs

Graphical definition of communication graphs has been a success of Poker. The class of graphs supported, however, has been a problem. In particular, Poker initial support for development of programs for the CHiP architecture led to Poker's $\mathcal{Y}$-level, which can only define communication graphs that are bounded to a constant, maximum degree (currently eight). Graphs that require higher connectivity, such as a binary 9-cube, cannot be handled easily in these bounded-degree graphs.

This restriction has, in practice, been a serious annoyance in the construction of some Poker programs. Potential solutions to this problem are to increase the maximum degree of the graph, which is at best a limited solution, or to support variable degree logical graphs. This second solution is attractive, but it is difficult to realize. The key problem with the solution is that mapping from the logical graphs to the target architectures is extremely difficult. We intend to try to handle these problems by a combination of automatic and programmer-directed techniques.

4 Conclusion

Poker started from the basic belief that visual support for explicit parallelism can qualitatively increase the productivity of parallel programmers. Our extensive experience now justifies this belief. For a variety of reasons–including changes in technology and the evolution of the implementation–the current Poker implementation falls short of an ideal environment in several key ways. In essence, Poker has improved the level at which we can program parallel computers, but we still are just entering the area of high-level parallel programming. Our experience positions us to build our next environment in way that will increase programmer productivity while ensuring that the programs produced are highly efficient.

Acknowledgments Poker would not have been possible without the great effort of people too numerous to list here. We also appreciate the comments and insights given to us by those who installed and used Poker at other sites.

References

[1] J.R. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. *Conference Record of the 14th Annual ACM Symposium on the Principles of Programming Languages* (January 1987).

[2] D.A. Bailey and J.E. Cuny. Graph Grammar Based Specification of Interconnection Structures for Massively Parallel Computation. In *Graph-Grammars and Their Applications to Computer Science*, H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld (editors). Lecture Notes in Computer Science 291, Springer-Verlag (1987).

[3] F.D. Berman and L. Snyder. On Mapping Parallel Algorithms into Parallel Architectures. *Proceedings of the International Conference on Parallel Processing* (1984).

[4] W.F. Crowley, C.P. Hendrickson, T.L. Rudy. The Simple Code. Technical Report UCID-17715, Lawrence Livermore Lab. (February 1978).

[5] A.K. Dewdney. Computer Recreations, *Scientific American* (December 1984).

[6] K. Gates and D. Socha. Programming N-cubes with a Graphical Parallel Programming Environment versus an Extended Sequential Language. In *Hypercube Multiprocessors 1987: Proceedings of the 2nd Conference on Hypercube Multiprocessors* (1987).

[7] R.B. Grafton and T. Ichikawa, editors. Special Issue on Visual Programming, *IEEE Computer 18*,8 (August 1985).

[8] T.J. Holman. An Evaluation of Floating Point Hardware in Parallel Computers. Technical Report 87-12-09, Department of Computer Science, University of Washington (December 1987).

[9] T.J. Holman and L. Snyder. A Transparent Coprocessor for Interprocessor Communication in an MIMD Computer. Technical Report 87-03-07, Department of Computer Science, University of Washington (March 1987).

[10] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe. The Structure of an Advanced Vectorizer for Pipelined Processors. *Proceedings of the 4th International Computer Software and Applications Conference* (1980).

[11] H.T. Kung and C.E. Leiserson. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, C. Mead and L. Conway. Addison-Wesley (1980).

[12] P.A. Nelson. A Non-Systolic Matrix Product Algorithm. Technical Report 85-11-02. Department of Computer Science, University of Washington (November 1985).

[13] P.A. Nelson. *Parallel Programming Paradigms.* PhD Thesis. Technical Report 87-07-02, Department of Computer Science, University of Washington (July 1987).

[14] J.T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems 2*,4 (October 1980).

[15] L. Snyder and D. Socha. Poker on the Cosmic Cube: The First Retargettable Parallel Programming Language and Environment. *Proceedings of the International Conference on Parallel Processing* (1986).

[16] L. Snyder. Type Architectures, Shared Memory and the Corollary of Modest Potential. In *Annual Review of Computer Science*, Volume 1 (1986).

[17] L. Snyder. Poker 4.0: A Programmer's Reference Guide. Technical Report 86-05-04, Department of Computer Science, University of Washington (1986).

[18] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer 17*,7 (July 1984).

[19] L. Snyder. Introduction to the Configurable Highly Parallel Computer. *IEEE Computer 15*,1 (January 1982).

[20] D. Socha, M. Bailey, and D. Notkin. Voyeur: Graphical Views of Parallel Programs. *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging* (May 1988).